Научная графика
лекция 3

по материалам:

https://metanit.com/web/react/2.1.php

Компоненты React

ReactElement плохо подходит для создания комплексной разметки HTML. Объекты ReactElement сложно использовать повторно в других аналогичных ситуациях, сложнее поддерживать.

И для решения этой проблемы в React используются компоненты. Компоненты проще обновлять и использовать повторно.

Компоненты аналогичны функциям JavaScript. Они хранят состояние с помощью свойств и возвращают элементы React, которые затем появляются на веб-странице.

Компоненты React

Компоненты можно определить различными способами. Первый способ - функциональный. Например:

```
1  function Hello() {
2    return <h1>Привет, Eugene</h1>;
3  }
```

Здесь определяется компонент Hello. Название компонентов должно начинаться с заглавной буквы.

Компоненты React

Второй способ определения компонентов предполагает использование классов ES6:

```
class Hello extends React.Component {
  render() {
    return <h1>Привет, Eugene</h1>;
  }
}
```

Для рендеринга компонента в классе компонента обязательно должен быть определен метод render(), который возвращает создаваемый элемент на JSX.

Компоненты React

Также для определения мы можем использовать стрелочные функции (arrow functions):

```
1  var Hello =() => {
2      return (<h1>Привет, Eugene</h1>);
3  }
```

Применение компонентов

Используем выше определенный компонент Hello

```
13    <script type="text/babel">
14        class Hello extends React.Component {
15            render() {
16              return <h1>Привет, Eugene</h1>;
17            }
18          }
19          ReactDOM.render(
20              <Hello />,
21              document.getElementById("app")
22          )
23    </script>
```

Props

Props представляет коллекцию значений, которые ассоциированы с компонентом.
Эти значения позволяют создавать динамические компоненты, которые не зависят от жестко закодированных статических данных.

Функциональный подход:

```
function Hello(props) {
  return <div>
            <p>Имя: {props.name}</p>
            <p>Возраст: {props.age}</p>
    </div>;
}
```

Параметр props, который передается в функцию компонента, инкапсулирует свойства объекта. В частности, свойство name и age.

При рендеринге мы можем создать набор компонентов Hello, но передать в них разные данные для name и age. И таким образом, получим набор однотипной разметки html, наполненной разными данными.

Props

Использование классов ES6:

```
class Hello extends React.Component {
  render() {
    return <div>
            <p>Имя: {this.props.name}</p>
            <p>Возраст: {this.props.age}</p>
    </div>;
  }
}
```

Класс компонента также извне получает объект свойств, который доступен через this.props.

Props

Использование классов ES6:

```
class Hello extends React.Component {
  render() {
    return <div>
            <p>Имя: {this.props.name}</p>
            <p>Возраст: {this.props.age}</p>
    </div>;
  }
}
```

Класс компонента также извне получает объект свойств, который доступен через this.props.

Props

Использование стрелочных функций (arrow functions):

```
const Hello = (props) => {

    const {name, age} = props;
    return(<div>
            <p>Имя: {name}</p>
            <p>Возраст: {age}</p>
           </div>);
}
```

Props

Используем компонент Hello:

```
13    <script type="text/babel">
14        class Hello extends React.Component {
15            render() {
16              return <div>
17                          <p>Имя: {this.props.name}</p>
18                          <p>Возраст: {this.props.age}</p>
19                      </div>;
20            }
21          }
22          ReactDOM.render(
23              <Hello name="Tom" age="33" />,
24              document.getElementById("app")
25          )
26    </script>
```

При рендеринге React передает значения атрибутов в виде единого объекта "props". То есть значение из атрибута name="Tom" перейдет в свойство props.name.

В итоге будет создана следующая страница:

Имя: Tom

Возраст: 33

Значения по умолчанию

Мы можем определить для свойств значения по умолчанию на тот случай, если их значения не передаются извне.
Если мы применяем классы, то для установки значений применяется статическое свойство defaultProps:

```
class Hello extends React.Component {

    render() {
        return <div>
            <p>Имя: {this.props.name}</p>
            <p>Возраст: {this.props.age}</p>
        </div>;
    }
}
Hello.defaultProps = {name: "Tom", age: 22};
```

Значения по умолчанию

При функциональном определении компонента также применяется свойство defaultProps:

```
function Hello(props) {
    return <div>
        <p>Имя: {props.name}</p>
        <p>Возраст: {props.age}</p>
    </div>;
}
Hello.defaultProps = {name: "Tom", age: 22};
```

Значения по умолчанию

И в любом из этих случаев, если мы не определим какие-то свойства для компонента, то они будут брать значения из значений по умолчанию:

```
13      <script type="text/babel">
14          class Hello extends React.Component {
15
16              render() {
17                  return <div>
18                      <p>Имя: {this.props.name}</p>
19                      <p>Возраст: {this.props.age}</p>
20                  </div>;
21              }
22          }
23          Hello.defaultProps = {name: "Tom", age: 22};
24          ReactDOM.render(
25              <Hello name="Bob" />,
26              document.getElementById("app")
27          )
28      </script>
```

Имя: Bob

Возраст: 22

Обновление props

Сам this.props представляет неизменяемый объект, который предназначен только для чтения. Поэтому не следует его изменять, например:

```
class Hello extends React.Component {
    render() {
        this.props.name = "Tom";
        return <h1>Привет, {this.props.name}</h1>;
    }
}
```

Если же надо время то времени изменять какие-то внутренние данные компонента, то для хранения таких данных в React предназначен объект **state**, который будет рассмотрен далее

События

Обработка событий элементов в React во многом похожа на обработку событий элементов DOM с помощью обычного JavaScript. В то же время есть небольшие отличия:
События в React используют camelCase (в стандартном html "onclick", в React - "onClick")

В JSX в обработчик события передается функция компонента, а не строка.

В React можно применять разные способы определения и вызова событий. Возьмем простейшую задачу - обработка нажатия кнопки.

События

Возьмем простейшую задачу - обработка нажатия кнопки. Распространенный подход заключается в установке привязки события в конструкторе компонента:

```
class ClickButton extends React.Component {

    constructor(props) {
        super(props);
        this.press = this.press.bind(this);
    }
    press(){
        console.log(this);
        alert("Hello React!")
    }
    render() {
        return <button onClick={this.press}>Click</button>;
    }
}
```

Здесь используется событие нажатия кнопки, которое задается через атрибут onClick (не onclick).

События

```
class ClickButton extends React.Component {

    constructor(props) {
        super(props);
        this.press = this.press.bind(this);
    }
    press(){
        console.log(this);
        alert("Hello React!")
    }
    render() {
        return <button onClick={this.press}>Click</button>;
    }
}
```

Этому атрибуту в качестве обработчика события передавалась функция this.press, которая определена в классе компонента. И при нажатии на кнопку будет вызываться функция press, которая с помощью функции alert отображает окно с некоторым уведомлением.

Главная сложность при использовании событий - это работа с ключевым словом this, которое указывает на текущий объект, в данном случае компонент.

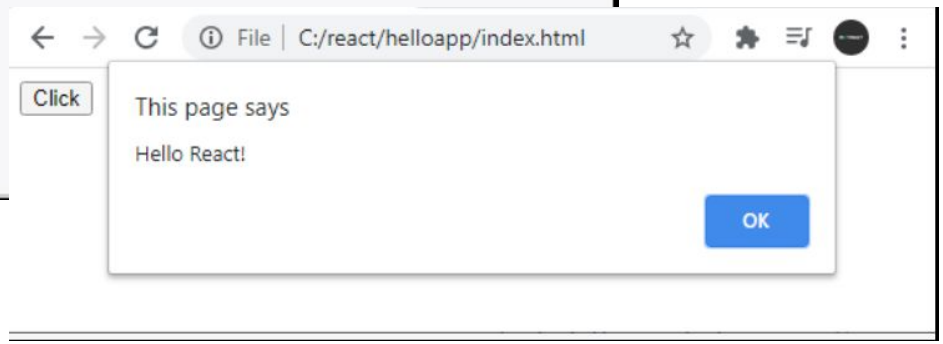
По умолчанию в функцию обработчика не передается текущий объект, поэтому this будет иметь значение undefined. И ни к каким свойствам и методам компонента через this мы обратиться не сможем. И чтобы в метод press корректно передавалась ссылка на текущий объект через this, в конструкторе класса прописывается вызов:

```
this.press = this.press.bind(this);
```

События

Используем компонент на веб-странице:

```
14          class ClickButton extends React.Component {
15
16              constructor(props) {
17                  super(props);
18
19                  this.press = this.press.bind(this);
20              }
21              press(){
22                  console.log(this);
23                  alert("Hello React!")
24              }
25              render() {
26                  return <button onClick={this.press}>Click</button>;
27              }
28          }
29
30          ReactDOM.render(
31              <ClickButton />,
32              document.getElementById("app")
33          )
```



События

Однако есть и другие способы определения и вызова события. Например, определение обработчика в виде публичного поля компонента, которое указывает на стрелочную функцию.

```
class ClickButton extends React.Component {

    press = () => {
        console.log(this);
        alert("Hello React!")
    }
    render() {
        return <button onClick={this.press}>Click</button>;
    }
}
```

События

Либо мы можем определить функцию обработчика события как обычный метод класса, а вызывать с помощью стрелочной функции:

```
class ClickButton extends React.Component {

    press(){
        console.log(this);
        alert("Hello React!");
    }
    render() {
        return <button onClick={() => this.press()}>Click</button>;
    }
}
```

Однако в случае с использованием стрелочной функции есть вероятность столкнуться с проблемой производительности, если функция обработчика передается через свойства **props** вложенным компонентам. Так как обработчик события будет создаваться каждый раз заново при каждом рендеринге компонента, что может привести к дополнительному повторному рендерингу вложенных компонентов, без которого можно было бы обойтись.

Поэтому использование конструктора является более предпочтительной практикой.

Получение информации о событии

React использует концепцию SyntheticEvent - специальных объектов, которые представляют собой обертки для объектов событий, передаваемых в функцию события. И используя такой объект, мы можем получить в обработчике события всю информацию о событии:

```
class ClickButton extends React.Component {

    constructor(props) {
        super(props);
        this.press = this.press.bind(this);
    }
    press(e){
        console.log(e); // выводим информацию о событии
        alert("Hello React!")
    }
    render() {
        return <button onclick="{this.press}">Click</button>;
    }
}
```

Параметр **e** - это и есть информация о событии, которая передается в обработчик системой и которую мы можем использовать при обработке.

Передача параметров в обработчик события

Если необходимо передать в обработчик события некоторые аргументы, то в этом случае можно вызвать обработчик через стрелочную функцию:

```
<script type="text/babel">
      class PrintButton extends React.Component {

         constructor(props) {
            super(props);

            this.print = this.print.bind(this);
         }
         print(name, age){
             console.log(`Name ${name}  Age: ${age}`);
         }
         render() {
            return <div>
                     <button onClick={() => this.print("Bob", 23)}>Print Bob</button>
                     <button onClick={() => this.print("Tom", 36)}>Print Tom</button>
                  </div>;
         }
      }

      ReactDOM.render(
         <PrintButton />,
         document.getElementById("app")
      )
   </script>
```

Передача параметров в обработчик события

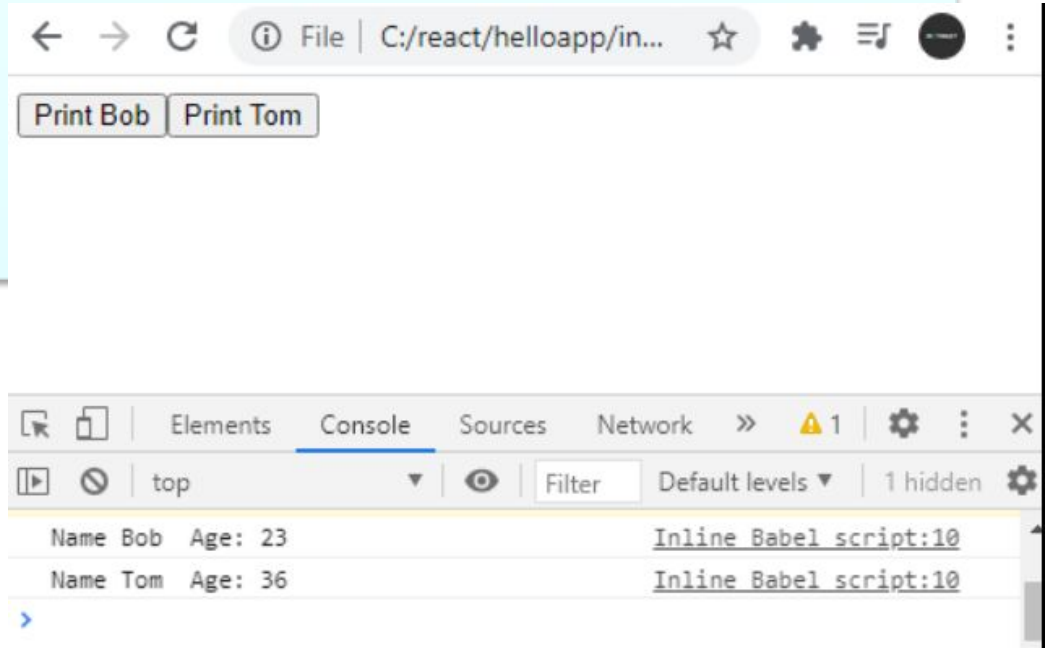
Если необходимо передать в обработчик события некоторые аргументы, то в этом случае можно вызвать обработчик через стрелочную функцию:

```
<script type="text/babel">
    class PrintButton extends React.Component {

        constructor(props) {
            super(props);

            this.print = this.print.bind(this);
        }
        print(name, age){
            console.log(`Name ${name}  Age: ${age}`);
        }
        render() {
            return <div>
                    <button onClick={() => this.print("Bob", 23)}>Print Bob</button>
                    <button onClick={() => this.print("Tom", 36)}>Print Tom</button>
                </div>;
        }
    }

    ReactDOM.render(
        <PrintButton />,
        document.getElementById("app")
    )
</script>
```



State

Объект **state** описывает внутреннее состояние компонента, он похож на props за тем исключением, что состояние определяется внутри компонента и доступно только из компонента.

Если props представляет входные данные, которые передаются в компонент извне, то состояние хранит такие объекты, которые создаются в компоненте и полностью зависят от компонента.

Также в отличие от **props** значения в **state** можно изменять.

И еще важный момент - значения из **state** должны использоваться при рендеринге. Если какой-то объект не используется в рендерниге компонента, то нет смысла сохранять его в **state**.

Нередко **state** описывает какие-то визуальные свойства элемента, которые могут изменяться при взаимодействие с пользователем.

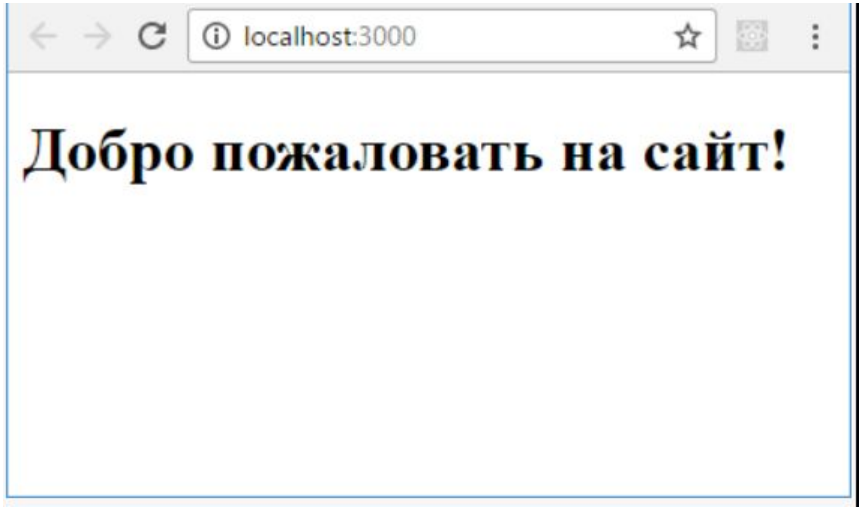
Например, кнопку нажали, и соответственно можно изменить ее состояние - придать ей какой-то другой цвет, тень и так далее. Кнопку нажали повторно - можно вернуть исходное состояние.

State

Единственное место, где можно установить объект **state** - это конструктор класса:

```
<script type="text/babel">
      class Hello extends React.Component {
         constructor(props) {
            super(props);
            this.state = {welcome: "Добро пожаловать на сайт!"};
         }
         render() {
            return <h1>{this.state.welcome}</h1>;
         }
      }
      ReactDOM.render(
         <Hello />,
         document.getElementById("app")
      )
   </script>
```

При определении конструктора компонента в нем должен вызываться конструктор базового класса, в который передается объект props.



Обновление состояния

Для обновления состояния вызывается функция setState():

```
this.setState({welcome: "Привет React"});
```

Изменение состояния вызовет повторный рендеринг компонента, в соответствии с чем веб-страница будет обновлена.
В то же время не стоит изменять свойства состояния напрямую, например:

```
this.state.welcome = "Привет React";
```

В данном случае изменения повторного рендеринга компонента происходить не будет.

При этом нам не обязательно обновлять все его значения. В процессе работы программы мы можем обновить только некоторые свойства. Тогда необновленные свойства будут сохранять старые значения.

Обновление состояния

```
<script type="text/babel">
      class ClickButton extends React.Component {

         constructor(props) {
            super(props);
            this.state = {class: "off", label: "Нажми"};

            this.press = this.press.bind(this);
         }
         press(){
            let className = (this.state.class==="off")?"on":"off";
            this.setState({class: className});
         }
         render() {
            return <button onClick={this.press} className={this.state.class}>{this.state.label}</button>;
         }
      }

      ReactDOM.render(
         <ClickButton />,
         document.getElementById("app")
      )
   </script>
```

Здесь определен компонент ClickButton, который по сути представляет кнопку. В состоянии кнопки хранится два свойства - надпись и класс. При нажатии на кнопку мы будем переключать с одного класса на другой. Событие нажатия кнопки через атрибут onClick связано с методом press(), в котором переключается класс кнопки.

При этом свойство state.label остается неизменным.

Асинхронное обновление

При наличии нескольких вызовов setState() React может объединять их в один общий пакет обновлений для увеличения производительности.

Так как объекты this.props и this.state могут обновляться асинхронно, не стоит полагаться на значения этих объектов для вычисления состояния. Например:

```
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

Для обновления надо использовать другую версию функции setState(), которая в качестве параметра принимает функцию. Данная функция имеет два параметра: предыдущее состояние объекта **state** и объект **props** на момент применения обновления:

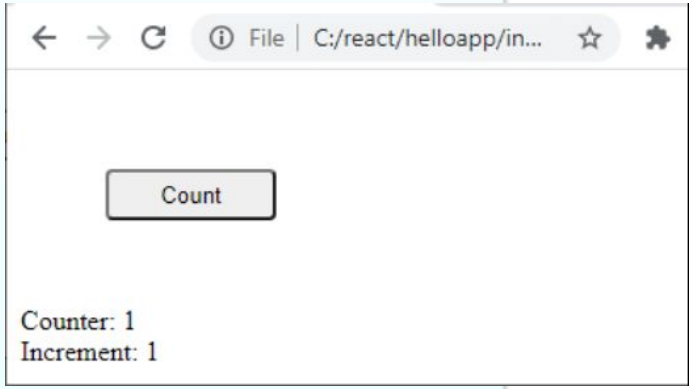
```
this.setState(function(prevState, props) {
  return {
    counter: prevState.counter + props.increment
  };
});
```

Асинхронное обновление

```
<script type="text/babel">
      class ClickButton extends React.Component {

        constructor(props) {
           super(props);
           this.state = {counter: 0};
           this.press = this.press.bind(this);
        }
        press(){
           this.setState({counter: this.state.counter + parseInt(this.props.increment)});
           this.setState({counter: this.state.counter + parseInt(this.props.increment)});
        }
        render() {
           return <div>
                     <button onClick={this.press}>Count</button>
                     <div>Counter: {this.state.counter} <br />Increment: {this.props.increment}</div>
                  </div>
        }
     }

     ReactDOM.render(
        <ClickButton increment="1" />,
        document.getElementById("app")
     )
   </script>
```



В props определено свойство increment - значение, на которое будет увеличиваться свойство counter в **state** *(this.setState({counter: this.state.counter + parseInt(this.props.increment)});)*.
При чем при нажатии кнопки мы предполагаем, что функция *setState()* будет вызываться два раза, соответственно значение *state.counter* при нажатии кнопки должно увеличиваться на 2. Однако в реальности увеличение происходит лишь на 1:

Асинхронное обновление

Теперь изменим код, применив второй вариант функции setState():

```
<script type="text/babel">
    class ClickButton extends React.Component {

      constructor(props) {
        super(props);
        this.state = {counter: 0};

        this.press = this.press.bind(this);
      }
      incrementCounter(prevState, props) {
          return {
            counter: prevState.counter + parseInt(props.increment)
          };
       }
      press(){
        this.setState(this.incrementCounter);
        this.setState(this.incrementCounter);
      }
      render() {
        return <div>
                <button onClick={this.press}>Count</button>
                <div>Counter: {this.state.counter}<br /> Increment: {this.props.increment}</div>
         </div>
      }
    }

    ReactDOM.render(
      <ClickButton increment="1" />,
      document.getElementById("app")
    )
  </script>
```

Чтобы избежать повторения, все действия по инкременту вынесены в отдельную функцию - incrementCounter, однако опять же функция setState() вызывается два раза. И теперь инкремент будет срабатывать два раза при однократном нажатии, собственно как и определено в коде и как и должно быть:

Жизненный цикл компонента

В процессе работы компонент проходит через ряд этапов жизненного цикла. На каждом из этапов вызывается определенная функция, в которой мы можем определить какие-либо действия:

1. **constructor(props)**: конструктор, в котором происходит начальная инициализация компонента

2. **static getDerivedStateFromProps(props, state)**: вызывается непосредственно перед рендерингом компонента. Этот метод не имеет доступа к текущему объекту компонента (то есть обратиться к объекту компоненту через this) и должен возвращать объект для обновления объекта state или значение null, если нечего обновлять.

3. **render()**: рендеринг компонента

4. **componentDidMount():** вызывается после рендеринга компонента. Здесь можно выполнять запросы к удаленным ресурсам

5. **componentWillUnmount():** вызывается перед удалением компонента из DOM

Жизненный цикл компонента

Кроме этих основных этапов или событий жизненного цикла, также имеется еще ряд функций, которые вызываются при обновлении состояния после начального рендеринга компонента, если в компоненте происходят обновления:

1. **static getDerivedStateFromProps(props, state)**

2. **shouldComponentUpdate(nextProps, nextState)**: вызывается каждый раз при обновлении объекта props или state. В качестве параметра передаются новый объект props и state. Эта функция должна возвращать true (надо делать обновление) или false (игнорировать обновление). По умолчанию возвращается true. Но если функция будет возвращать false, то тем самым мы отключим обновление компонента, а последующие функции не будут срабатывать.

3. **render()**: рендеринг компонента (если shouldComponentUpdate возвращает true)

4. **getSnapshotBeforeUpdate(prevProps, prevState)**: вызывается непосредственно перед компонента. Он позволяет компоненту получить информацию из DOM перед возможным обновлением. Возвращает в качестве значения какой-то отдельный аспект, который передается в качестве третьего параметра в метод componentDidUpdate() и может учитываться в componentDidUpdate при обновлении. Если нечего возвращать, то возвращается значение null

5. **componentDidUpdate(prevProps, prevState, snapshot)**: вызывается сразу после обновления компонента (если shouldComponentUpdate возвращает true). В качестве параметров передаются старые значения объектов props и state. Третий параметр - значение, которое возвращает метод getSnapshotBeforeUpdate

Жизненный цикл компонента

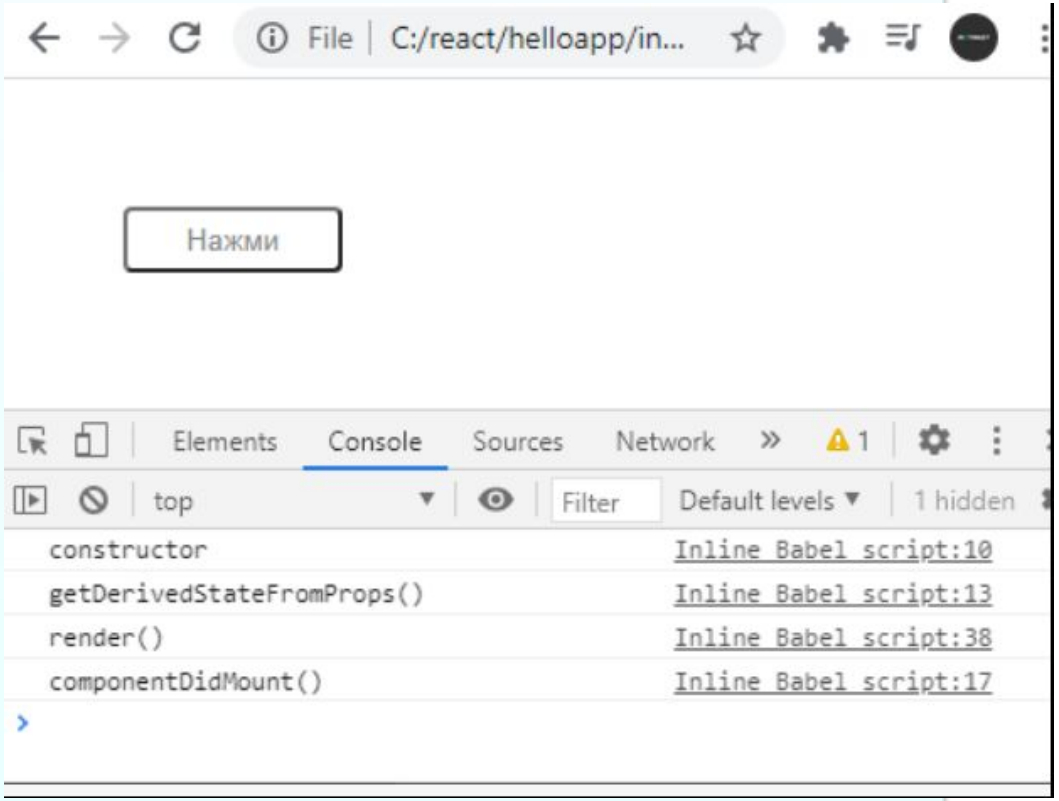
Применим некоторые из событий жизненного цикла:

```
<script type="text/babel">
    class ClickButton extends React.Component {

        constructor(props) {
            super(props);
            this.state = {class: "off", label: "Нажми"};

            this.press = this.press.bind(this);

            console.log("constructor");
        }
        static getDerivedStateFromProps(props, state) {
            console.log("getDerivedStateFromProps()");
            return null;
        }
        componentDidMount(){
            console.log("componentDidMount()");
        }
        componentWillUnmount(){
            console.log("componentWillUnmount()");
        }
        shouldComponentUpdate(){
            console.log("shouldComponentUpdate()");
            return true;
        }
        getSnapshotBeforeUpdate(prevProps, prevState) {
            console.log("getSnapshotBeforeUpdate()");
            return null;
        }
        componentDidUpdate(){
            console.log("componentDidUpdate()");
        }
        press(){
            var className = (this.state.class==="off")?"on":"off";
            this.setState({class: className});
        }
        render() {
            console.log("render()");
            return <button onClick={this.press} className={this.state.class}>{this.state.label}</button>;
        }
    }
    ReactDOM.render(
        <ClickButton />,
        document.getElementById("app")
    )
</script>
```



Жизненный цикл компонента

При нажатии на кнопку сработает обработчик нажатия, который обновит объект state, что вызовет еще ряд функций жизненного цикла и повторный рендеринг: